

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра математического обеспечения и применения ЭВМ**

**КУРСОВАЯ РАБОТА**  
**по дисциплине «Алгоритмы и структуры данных»**  
**Тема: Рандомизированное БДП**

Студент гр. 8304

\_\_\_\_\_

Ястребов И.М.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2019

**ЗАДАНИЕ  
НА КУРСОВУЮ РАБОТУ**

Студент Ястребов И.М.

Группа 8304

Тема работы: Рандомизированное БДП

Исходные данные: необходимо провести исследование алгоритма вставки и удаления в рандомизированное бинарное дерево поиска, включающее генерацию входных данных, использование их для измерения количественных характеристик алгоритмов, сравнение экспериментальных результатов с теоретическими.

Содержание пояснительной записки:

«Содержание», «Введение», «Задание», «Описание программы», «Тестирование», «Исследование», «Заключение», «Список использованных источников».

Предполагаемый объем пояснительной записки:

Не менее 30 страниц.

Дата выдачи задания:

Дата сдачи реферата:

Дата защиты реферата:

Студент		Ястребов И.М.
Преподаватель		Фирсов М.А.

## **АННОТАЦИЯ**

В ходе выполнения курсовой работы была разработана программа с GUI, позволяющая исследовать алгоритм вставки в рандомизированное бинарное дерево поиска, а также удаление заданного элемента. Программа обладает следующей функциональностью: построение РБДП по входному файлу, работа с пользовательской консолью, генерация тестов, вывод результатов тестов.

## **SUMMARY**

In the course work a program with a GUI was developed that allows you to examine the algorithm for inserting into a randomized binary search tree, as well as deleting a given element. The program has the following functionality: building a Treap from an input file, UI, test generating, test results output.

## СОДЕРЖАНИЕ

	Введение	5
1.	Задание	6
2.	Описание программы	7
2.1.	Описание основного класса для рандомизированных бдп	7
2.2.	Описание алгоритма вставки и удаления в рандомизированном бдп	8
2.3.	Описание генерирования входных значений	8
3	Тестирование	9
3.1.	Вид программы	9
4	Исследование	9
4.1	План экспериментального исследования	9
4.2	Исследование зависимостей от полученной высоты дерева для алгоритма вставки	10
4.3	Исследование зависимостей от количества итераций для алгоритма вставки	12
4.4	Исследование зависимостей от количества итераций для алгоритма удаления	13
4.5	Выводы об исследовании алгоритма	15
	Заключение	16
	Список использованных источников	17
	Приложение А. Исходный код программы. LAB5.CPP	18
	Приложение Б. Исходный код программы. TREAP.HPP	22

## ВВЕДЕНИЕ

### Цель работы

Реализация и экспериментальное машинное исследование алгоритмов работы с рандомизированными бинарными деревьями поиска.

### Основные задачи

Генерация входных данных, использование их для измерения количественных характеристик структур данных, алгоритмов, действий, сравнение экспериментальных результатов с теоретическими.

### Методы решения

Разработка программы велась на базе операционной системы Windows 10 в среде разработки MSVS 2019. Язык C++ стандарта 2017 года.



## 1. ЗАДАНИЕ

Необходимо провести исследование алгоритма вставки и удаления в рандомизированном бинарном дереве поиска в среднем и худшем случаях.

Исследование должно содержать:

1. Анализ задачи, цели, технологию проведения и план экспериментального исследования.
2. Генерацию представительного множества реализаций входных данных (с заданными особенностями распределения (для среднего и для худшего случаев)).
3. Выполнение исследуемых алгоритмов на сгенерированных наборах данных. При этом в ходе вычислительного процесса фиксируется как характеристики (например, время) работы программы, так и количество произведенных базовых операций алгоритма.
4. Фиксацию результатов испытаний алгоритма, накопление статистики.
5. Представление результатов испытаний, их интерпретацию и сопоставление с теоретическими оценками.

## 2. ОПИСАНИЕ ПРОГРАММЫ

### 2.1. Описание основного класса для рандомизированных бдп

Для реализации бдп был создан шаблонный класс `Node<typename Elem, typename Priority>`. Основные методы класса представлены в табл. 5.

Таблица 5 – Основные функции работы с декартовым деревом

Функция	Назначение
<code>Node() = default;</code>	Конструктор по умолчанию
<code>~Node() = default;</code>	Деструктор по умолчанию
<code>Node(elem key, priority prior) : key(key), prior(prior), left(nullptr), right(nullptr) { }</code>	Перегрузка конструктора
<code>static void split(nodePtr&lt;elem, priority&gt;, elem, nodePtr&lt;elem, priority&gt; &amp;, nodePtr&lt;elem, priority&gt; &amp;);</code>	Разбиение по ключу
<code>static void insert(nodePtr&lt;elem, priority&gt; &amp;, nodePtr&lt;elem, priority&gt;);</code>	Вставка нового элемента
<code>static void merge(nodePtr&lt;elem, priority&gt; &amp;, nodePtr&lt;elem, priority&gt;, nodePtr&lt;elem, priority&gt;);</code>	Слияние двух РБДП
<code>static bool erase(nodePtr&lt;elem, priority&gt; &amp;, elem);</code>	Удаление элемента
<code>tatic bool search(nodePtr&lt;elem, priority&gt; &amp;, elem);</code>	Поиск элемента
<code>static int depth(nodePtr&lt;elem, priority&gt; &amp;);</code>	Нахождение глубины РБДП

Программа имеет возможность отображения дерева в виде КЛП-скобочной записи, однако при тестировании эта функция использовалась лишь на малых объемах данных.

## 2.2. Описание алгоритма вставки и удаления в рандомизированном бдп

Известно, что если заранее перемешать как следует все ключи и потом построить из них дерево (ключи вставляются по стандартной схеме в полученном после перемешивания порядке), то построенное дерево окажется неплохо сбалансированным (его высота будет порядка  $2\log_2 n$  против  $\log_2 n$  для идеально сбалансированного дерева). Любой вводимый ключ может оказаться корнем с вероятностью  $\frac{1}{n+1}$  ( $n$  — размер дерева до вставки), следовательно выполняется с указанной вероятностью вставка в корень, а с вероятностью  $1 - \frac{1}{n+1}$  — рекурсивную вставку в правое или левое поддерево в зависимости от значения ключа в корне.

Удаление происходит по ключу — ищется узел с заданным ключом и этот узел удаляется из дерева. Основное свойство дерева поиска — любой ключ в левом поддереве меньше корневого ключа, а в правом поддереве — больше корневого ключа. Это свойство позволяет очень просто организовать поиск заданного ключа, перемещаясь от корня вправо или влево в зависимости от значения корневого ключа. Далее происходит объединение левого и правого поддеревьев найденного узла, удаляется узел и возвращается корень объединенного дерева.

## 2.3. Описание генерирования входных значений

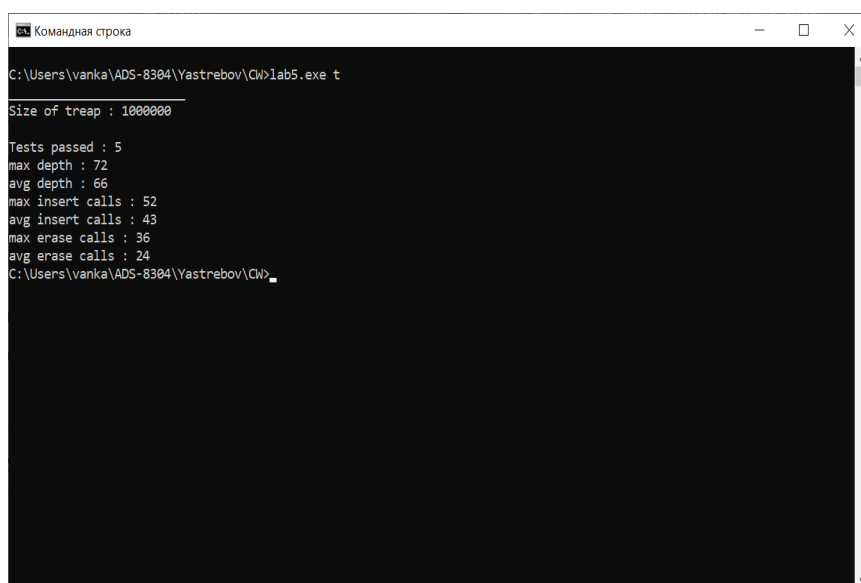
На каждой итерации тестирования выбранному количеству ключей псевдослучайным образом выдаются приоритеты. В качестве ключей используется набор `[1..TEST_SIZE]`, поскольку двойное перемешивание статистически эквивалентно одиночному.



### 3. ТЕСТИРОВАНИЕ

#### 3.1. Вид программы

Вид программы после запуска представлен на рис. 2.



```
Командная строка
C:\Users\vanka\ADS-8304\Yastrebov\CW\lab5.exe t
Size of treap : 1000000
Tests passed : 5
max depth : 72
avg depth : 66
max insert calls : 52
avg insert calls : 43
max erase calls : 36
avg erase calls : 24
C:\Users\vanka\ADS-8304\Yastrebov\CW>
```

Рисунок 2 – Вид программы после запуска

### 4. ИССЛЕДОВАНИЕ

#### 4.1. План экспериментального исследования.

Для проведения исследования сложности алгоритма вставки в рандомизированное бдп необходимо понимать, что на одном и том же наборе данных при повторном запуске будут получаться различные бдп. Опорным элементом при исследовании алгоритма станет высота получаемого дерева и число совершенных итераций. По аналогичному плану будет проводиться и исследование алгоритма удаления. После накопления данных необходимо провести сравнение результатов и сделать выводы об эффективности алгоритмов. Кроме того, объединив всю статистику, следует сравнить полученные экспериментальные зависимости от теоретических и сделать выводы о сложности алгоритма вставки в среднем и худшем случаях.

План проведения исследования:

- Получение информации о зависимости высоты дерева и числа итераций
- Анализ собранной информации, выводы о зависимостях эффективности алгоритма от указанных параметров
- Анализ собранной информации, сравнение экспериментальных значений с теоретическими, выводы о сложности алгоритма вставки и удаления.

#### 4.2. Исследование зависимостей от полученной высоты дерева для алгоритма вставки

Так как при построении бдп приоритеты выдаются псевдослучайным образом, то при одних и тех же значениях ключей полученные деревья будут различаться. Был проведён ряд тестов: на каждой итерации тестирования для набора ключей  $\leq 1..1e6$  псевдослучайным образом выбирались приоритеты — после чего происходили вызовы операций вставки новых и удаления существующих элементов.

С учетом того, максимальный размер данных выбран порядка  $1e6$ , что помогает более наглядно увидеть асимптотику алгоритма, а данные (ключи) не выбирались случайным образом и были описаны выше, далее они приведены не будут. Также, с учетом большого количества тестов, далее будут представлены только результаты экспериментов. Теоретическая функция -  $c \log_2 n$ . В скобках указаны результаты эксперимента при смоделированном вырожденном случае. На графике эти точки будут далеко за областью видимости, но хорошо видно, что они имеют порядок количества ключей — то есть в худшем случае РБДП работает за  $O(n)$ .

Количество ключей	Максимальная высота	Средняя высота
10	5(10)	5(10)
100	16(100)	13(100)
1000	26(1000)	22(1000)
10000	32(10000)	30(10000)
100000	42(100000)	40(100000)
1000000	70(1000000)	66(1000000)

Далее был построен сравнительный график, где представлены зависимости максимальной высоты от количества ключей, минимальной высоты от количества ключей, а так же теоретическая функция  $c \log_2 n$ . Полученные данные представлены на рис. 4.

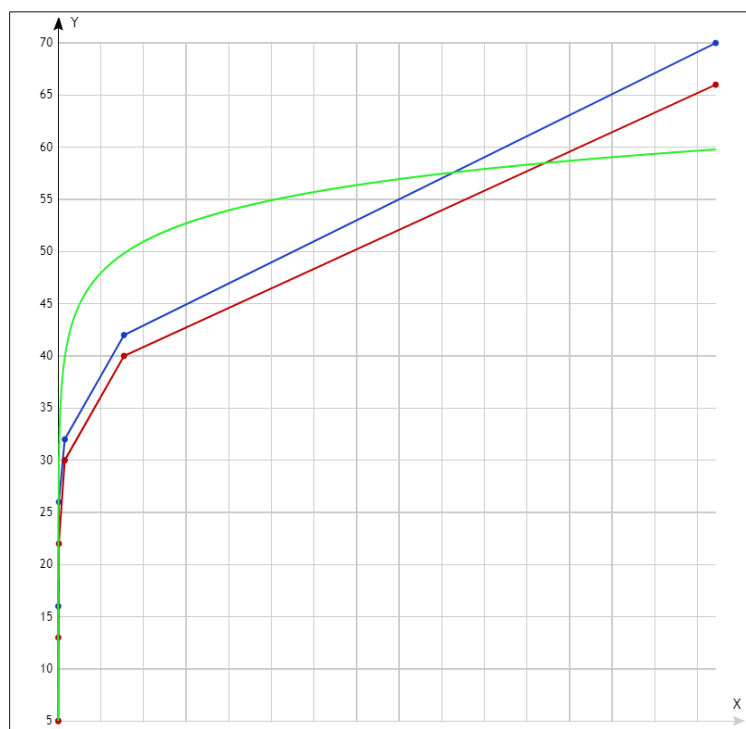


Рисунок 4 – Результаты тестирования 8 относительно теоретических значений

Исходя из полученных результатов можно увидеть, что глубина дерева является логарифмической. Стоит отметить, что худший случай, когда РБДП вырождается в линейный список с асимптотикой  $O(n)$  на все операции,

сгенерировать самостоятельно не получится, так как алгоритм использует рандомизацию для определения вероятности вставки в корень и, далее, высота дерева фиксируется. Это приводит к тому, что вероятность получения несбалансированного дерева оказывается пренебрежимо малой при больших размерах деревьев.

#### 4.3. Исследование зависимостей от количества итераций для алгоритма вставки

Был проведен ряд тестов, где в уже существующее бинарное дерево различного размера добавлялся новый элемент с псевдослучайно выданным приоритетом. В скобках указаны результаты эксперимента при смоделированном вырожденном случае. На графике эти точки будут далеко за областью видимости, но хорошо видно, что они имеют порядок количества ключей — то есть в худшем случае РБДП работает за  $O(n)$ .

Ниже представлены табл. 16 и рис.8, иллюстрирующие полученные результаты.

Таблица 16 – Результаты тестирования

Размер массива	Среднее арифметическое число итераций	Максимальное число итераций
10	4(7)	7(12)
100	8(64)	12(98)
1000	9(560)	13(863)
10000	14	16
100000	15	20
1000000	39	46

Продолжение таблицы 16

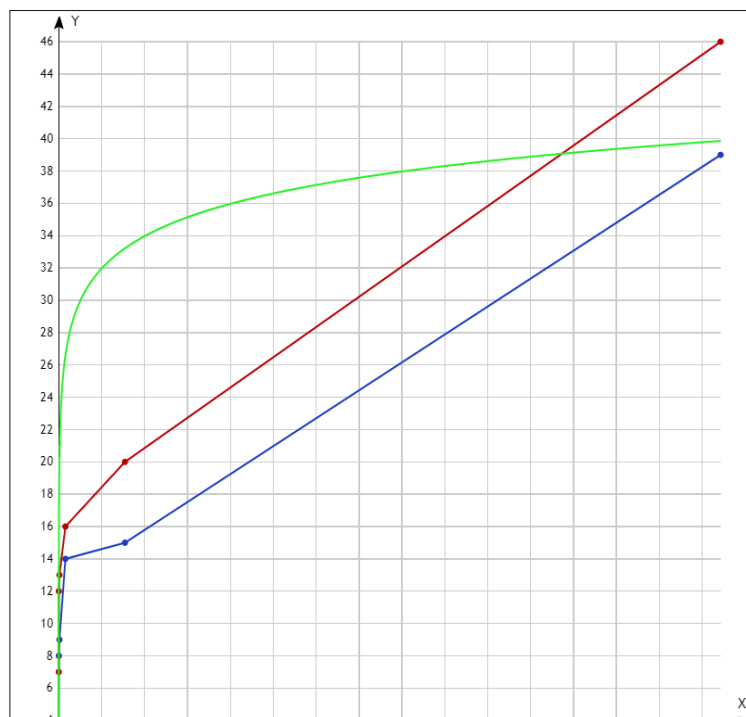


Рисунок 8 – График зависимости количества итераций от размера исходных данных

Количество итераций необходимое для вставки одного элемента в дерево составляет  $\log_2 n$ . Учитывая, что дерево имеет размер  $n$ , итоговая сложность алгоритма вставки для всего дерева составит  $n \cdot \log_2 n$

В целом можно сделать вывод, что алгоритм вставки в рандомизированном бинарном дереве поиска является крайне эффективным и с большой вероятностью строит сбалансированное дерево, избегая самого худшего случая – вырождения дерева в односвязный список.

#### 4.4. Исследование зависимостей от количества итераций для алгоритма удаления

Для проведения тестов генерировался индекс числа, которое требовалось удалить и затем запускался ряд тестов для одного и того же набора значений. Так как генерируемое каждый раз дерево получалось различным, выбранный элемент также менял своё местоположение. На основе этих данных была сведена таблица 17, где указано наибольшее, наименьшее и среднее арифметическое число итераций.

Таблица 17 – Результаты тестирования

Размер массива	Среднее арифметическое число итераций	Максимальное число итераций
10	5(8)	6(9)
100	10(35)	14(76)
1000	13(480)	16(743)
10000	19	23
100000	24	31
1000000	22	28

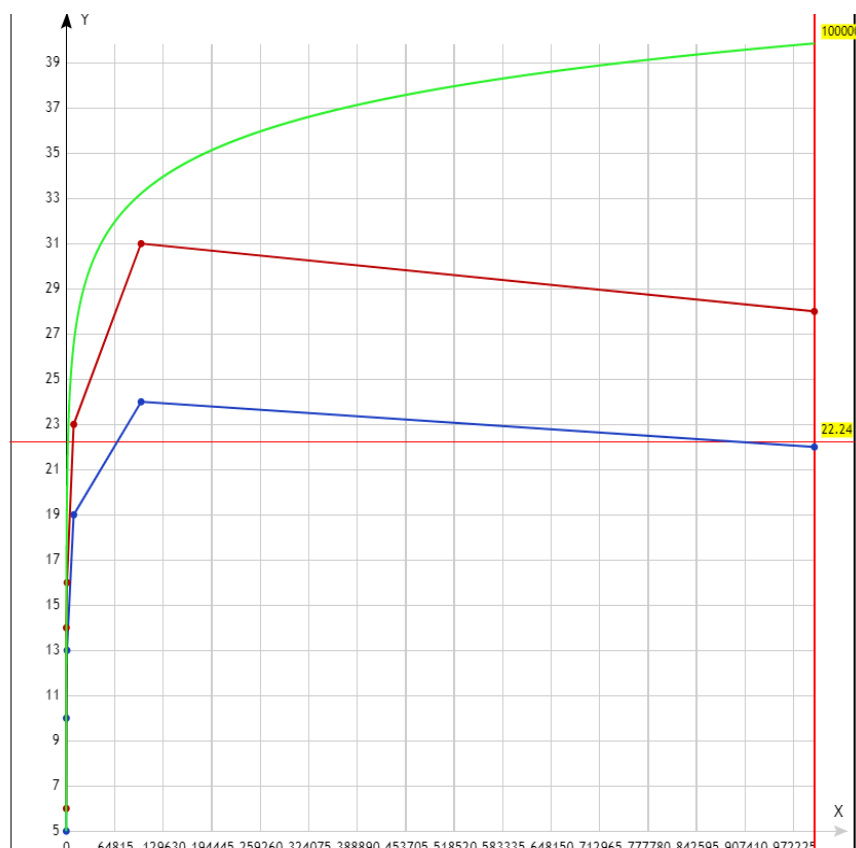


Рисунок 9 – График зависимости количества итераций от размера исходных данных

В результате можно сделать вывод, что несмотря на довольно резкие скачки получаемых значений (происходящие из-за случайной генерации удаляемого числа и повторного запуска теста для этого же набора данных: что

меняет его положение в дереве), алгоритм удаления одного элемента из бинарного дерева поиска обладает логарифмической сложностью.

#### **4.5. Выводы об исследовании алгоритма**

В результате исследования было подтверждено, что средняя сложность алгоритмов вставки и удаления в рандомизированном бинарном дереве поиска не превышает логарифмическую. Использование такого бинарного дерева поиска позволит с огромной вероятностью избежать полностью несбалансированных случаев, т.е. вырождения дерева в односвязный список.

Одним из основополагающих моментов эффективного поиска в таком бинарном дереве (для дальнейшего удаления элемента, к примеру) является правило хранения ключей: любой ключ в левом поддереве меньше корневого ключа, а в правом поддереве — больше корневого ключа.

Следует отметить, что значительную роль в построении такого дерева играет генерация вероятностей: именно от неё зависит наиболее оптимальная высота получаемого дерева.

## **ЗАКЛЮЧЕНИЕ**

В ходе выполнения курсовой работы была разработана программа, которая обладает следующей функциональностью: построение РБПД по входному файлу, генерация тестов, вывод результатов тестов, работа с пользовательской консолью. С помощью программы было проведено исследование различных случаев алгоритма вставки в рандомизированное бинарное дерево и алгоритма удаления заданного значения. В ходе исследования была выявлена зависимость эффективности алгоритма от различных параметров. В результате было выявлено, что на эффективность влияет распределение приоритетов по ключам. В среднем случае дерево получается хорошо сбалансированным, особенно на больших объёмах данных.



#### СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Bjarne Stroustrup. A Tour of C++. М.: Addison-Wesley, 2018. 217 с.
2. Treap // GeeksforGeeks <https://www.geeksforgeeks.org/treap-a-random-ized-binary-search-tree/> (дата обращения: 18.12.2000)
3. Qt Documentation // Qt. URL: <https://doc.qt.io/qt-5/index.html> (дата обращения: 18.12.2000)
4. Рандомизированные деревья поиска URL: <https://habr.com/ru/post/145388/#reed> (дата обращения: 17.12.2000)
5. The Height of a Random Binary Search Tree // BRUCE REED URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.1289&rep=rep1&type=pdf> (дата обращения: 18.12.2000)

**ПРИЛОЖЕНИЕ А**  
**ИСХОДНЫЙ КОД ПРОГРАММЫ. LAB5.CPP**

```
#include <iostream>
#include <fstream>
#include "Treap.hpp"
#include <vector>
#include <string>
#include <ctime>
#include <sstream>
#include <time.h>
#include <ctgmath>
#define TEST_SIZE 1000000
#define TEST_COUNT 5

template<typename elem, typename priority>
void printtree(nodePtr<elem, priority>& head) {
    if (!head) {
        std::cout << '#';
        return;
    }
    std::cout << '(';
    std::cout << "(" << head->key << ";" << head->prior << ")";
    printtree(head->left);
    printtree(head->right);
    std::cout << ')';
}

int main(int argc, char* argv[]) {
    bool test(false);

    if ((argc == 2) && (*(argv[1]) == 't'))
        test = true;

    std::streambuf* backup;
    backup = std::cin.rdbuf();

    std::ifstream ifs;

    ifs.open("input.txt");

    std::cin.rdbuf(ifs.rdbuf());
```

```

nodePtr<int, int> head = nullptr;

int value(0);

srand((unsigned int)time(0));

if(!test)
while (std::cin >> value)
{
    if (!Node<int, int>::search(head, value))
        Node<int, int>::insert(head, std::make_shared<Node<int,
int>>(value, rand() % INT_MAX));
    else
        std::cout << "Already there" << std::endl;
}

std::cin.rdbuf(backup);

if (!test) {
    std::cout << "tree : " << std::endl;

    printtree(head);
    std::cout << std::endl;
}

for(bool f(1); f && !test; ) {
    std::cout << "Choose action :\n1 - insert\n2 - erase\n3 - exit" <<
std::endl;

    int action(0);
    std::cin >> action;

    switch (action) {
    case 1:
        std::cout << "Enter value" << std::endl;

        std::cin >> value;

        if (!Node<int, int>::search(head, value))
            Node<int, int>::insert(head, std::make_shared<Node<int,
int>>(value, rand() % INT_MAX));
        else
            std::cout << "Already there" << std::endl;

        printtree(head);

```

```

        std::cout << std::endl;

        break;

    case 2:
        std::cout << "Enter value" << std::endl;

        std::cin >> value;

        Node<int, int>::erase(head, value);

        printtree(head);
        std::cout << std::endl;

        break;

    case 3:
        return 0;

    default:
        f = !f;
        break;
}
}

```

```

for (int size = 10; size <= TEST_SIZE; size *= 10) {
    int maxDepth(0), avgDepth(0), maxInsertCalls(0), avgInsertCalls(0),
    maxEraseCalls(0), avgEraseCalls(0);

    std::cout << "\n_____ \n";
    std::cout << "Size of treap : " << size << std::endl;

    for (int j = 0; j < TEST_COUNT; j++) {
        nodePtr<int, int> testHead = nullptr;

        for (int i = 0; i < size; ++i)
        {
            if (!Node<int, int>::search(testHead, i))
                Node<int, int>::insert(testHead,
std::make_shared<Node<int, int>>(i, rand() % INT_MAX));
            else
                std::cout << "Already there" << std::endl;
        }
    }
}

```

```

        int tmp = Node<int, int>::depth(testHead);
        avgDepth += tmp;

        if (tmp > maxDepth)
            maxDepth = tmp;

        counter = 0;

        Node<int, int>::insert(testHead, std::make_shared<Node<int,
int>>(size + 1, rand() % INT_MAX));

        avgInsertCalls += counter;

        if (counter > maxInsertCalls)
            maxInsertCalls = counter;

        counter = 0;

        ///

        Node<int, int>::erase(testHead, rand() % size);

        avgEraseCalls += counter;

        if (counter > maxEraseCalls)
            maxEraseCalls = counter;

        counter = 0;

    }

    std::cout << "\nTests passed : " << TEST_COUNT << "\nmax depth : "
<< maxDepth << "\navg depth : " << avgDepth / TEST_COUNT << "\nmax insert calls
: " << maxInsertCalls << "\navg insert calls : " << avgInsertCalls / TEST_COUNT
<< "\nmax erase calls : " << maxEraseCalls << "\navg erase
calls : " << avgEraseCalls / TEST_COUNT;
    }

    return 0;
}

```

**ПРИЛОЖЕНИЕ Б**  
**ИСХОДНЫЙ КОД ПРОГРАММЫ. TREAP.HPP**

```
#pragma once
#include <memory>
#include <algorithm>

static int counter = 0;

template<typename elem, typename priority>
class Node;

template<typename elem, typename priority>
using nodePtr = std::shared_ptr<Node<elem, priority>>;

template<typename elem, typename priority>
class Node {
public:
    elem key;
    priority prior;

    nodePtr<elem, priority> left, right;

    Node() = default;

    //default copy constructor and operator= are intended

    ~Node() = default;

    Node(elem key, priority prior) : key(key), prior(prior), left(nullptr),
right(nullptr) { }

    static void split(nodePtr<elem, priority>, elem, nodePtr<elem, priority>
&, nodePtr<elem, priority> &);

    static void insert(nodePtr<elem, priority> &, nodePtr<elem, priority>);

    static void merge(nodePtr<elem, priority> &, nodePtr<elem, priority>,
nodePtr<elem, priority>);

    static bool erase(nodePtr<elem, priority> &, elem);

    static bool search(nodePtr<elem, priority> &, elem);

    static int depth(nodePtr<elem, priority> &);
};

template<typename elem, typename priority>
bool Node<elem, priority>::search(nodePtr<elem, priority>& head, elem key)
{
    ++counter;

    if (!head)
        return false;

    if (head->key == key)
        return true;

    return search(head->key < key ? head->right : head->left, key);
}
```

```

template<typename elem, typename priority>
void Node<elem, priority>::split(nodePtr<elem, priority> head, elem key,
nodePtr<elem, priority> & left, nodePtr<elem, priority> & right) {
    ++counter;

    if (!head) {
        left = nullptr;
        right = nullptr;
    }

    else if (key < head->key) {
        split(head->left, key, left, head->left);
        right = head;
    }

    else {
        split(head->right, key, head->right, right);
        left = head;
    }
}

template<typename elem, typename priority>
void Node<elem, priority>::insert(nodePtr<elem, priority> &t, nodePtr<elem,
priority> it) {
    ++counter;

    if (!t)
        t = it;

    else if (it->prior > t->prior) {
        split(t, it->key, it->left, it->right);
        t = it;
    }

    else
        insert(it->key < t->key ? t->left : t->right, it);
}

template<typename elem, typename priority>
void Node<elem, priority>::merge(nodePtr<elem, priority> & t, nodePtr<elem,
priority> l, nodePtr<elem, priority> r) {
    ++counter;

    if (!l || !r)
        t = l ? l : r;

    else if (l->prior > r->prior) {
        merge(l->right, l->right, r);
        t = l;
    }

    else {
        merge(r->left, l, r->left);
        t = r;
    }
}

template<typename elem, typename priority>

```

```

bool Node<elem, priority>::erase(nodePtr<elem, priority> & t, elem key) {
    ++counter;

    if (!t)
        return false;

    if (t->key == key) {
        merge(t, t->left, t->right);

        return true;
    }

    return erase(key < t->key ? t->left : t->right, key);
}

template<typename elem, typename priority>
int Node<elem, priority>::depth(nodePtr<elem, priority> &head)
{
    if (!head)
        return 0;

    return std::max(depth(head->left), depth(head->right)) + 1;
}

```



